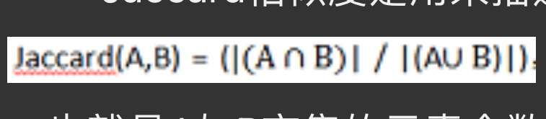
在数据挖掘中，有一个比较基本的问题，就是比较两个集合的相似度。关于这个问题，最笨的方法就是用一个两重循环来遍历这两个集合中的所有元素，进而统计这两个集合中相同元素的个数。但是，当这两个集合里的元素数量非常庞大时，同时又有很多个集合需要判断两两之间的相似度时，这种方法就呵呵了，对内存和时间的消耗都非常大。因此，为了解决这个问题，数据挖掘中有另一个方法。

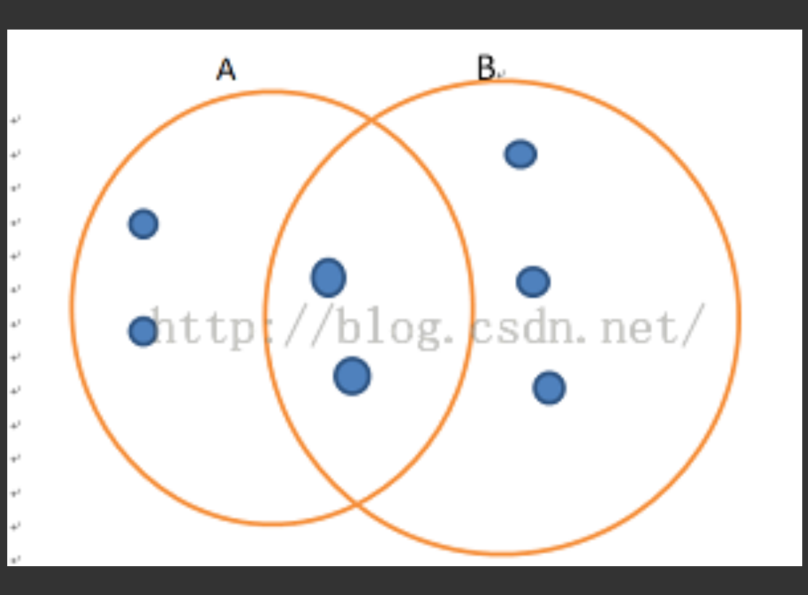
Jaccard相似度

在介绍具体算法之前，我们首先来了解一个概念：Jaccard相似度。

Jaccard相似度是用来描述两个集合间的相似度的，其计算方法如下（假设有两个集合A，B）：


Jaccard(A,B) = (|(A ∩ B)| / |(A∪ B)|)

，也就是A与B交集的元素个数除以A与B并集的元素个数；为了书写方便，下面的讨论中我们将集合A和B的Jaccard相似度记为SIM(A,B)；



例如：上图中有两个集合A，B；A中有4个元素，B中有5个元素；A，B的交集元素个数为2，并集元素个数为7，所以SIM(A,B) = 2 / 7；

k-Shingle

假如我们把一整篇文章看成一个长的字符串，那么k-shingle就是这篇文档中长度为k的任意字符子串。所以，一篇文章就是很多个不同的k-shingle的集合。

例如：现在我们有一篇很短的文章，文章内容为abcdabd，令k=2，那么这篇文章中所有的2-shingle组成的集合为{ab,bc,cd,da,bd}，需要注意的是，ab在文章中出现了两次，但是在集合中只出现一次，这是因为集合中不能有相同的元素。

尽管用k-shingle的方式来表示每篇文章，然后再通过判断每篇文章中shingle集合的相同元素的数量，就可以得出文章的相似度；但是，一篇文章得到的shingle集合的元素个数是很多的。假定k=4，那么每个shingle中就会有4个字符，存在内存中就至少需要4个字节；那么要以这种方式存下一篇文章的所有shingle，需要的内存空间大概是原文档大小的4倍（假设原文档大小为100K，那么存下这篇文档的所有shingle则需要400K），这是因为原文档中的每个字符串都会出现在4个shingle中（除了开头和结尾那几个字符）。因此，以shingle的方式来存文章会消耗大量的内存。

接下来，我们需要把上面的shingle集合替换成规模小很多的“签名”集合。这样，就可以通过比较两篇文章的签名集合的相似度，就能够估计shingle的相似度了。这样得到的相似度只是原来相似度的近似值。

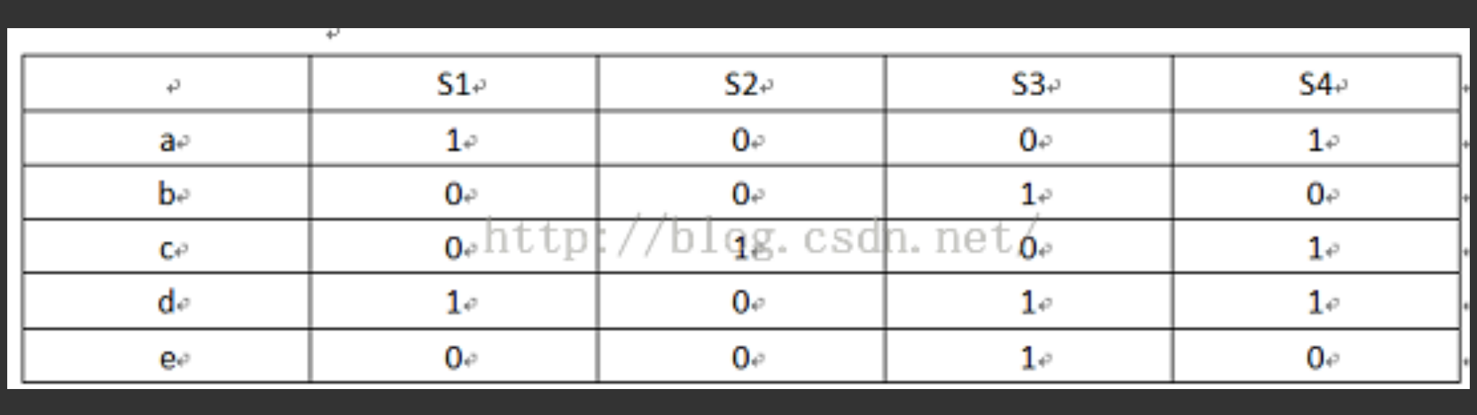
在介绍签名集合之前，我们先来看下如何将集合表示成其特征矩阵。

特征矩阵

特征矩阵的一列就对应一个集合，所有的行加起来就是所有集合元素的全集，如果集合中有那个元素，则矩阵中的对应位置为1，否则为0（好吧，讲的不是很清楚，还是直接上例子吧）：

假设现在有4个集合，分别为S1，S2，S3，S4；其中，S1={a,d}, S2={c}, S3={b,d,e}, S4={a,c,d}，所以全集U={a,b,c,d,e}

那么这4个集合的特征矩阵为：



	S1	S2	S3	S4
a	1	0	0	1
b	0	0	1	0
c	0	1	0	1
d	1	0	1	1
e	0	0	1	0

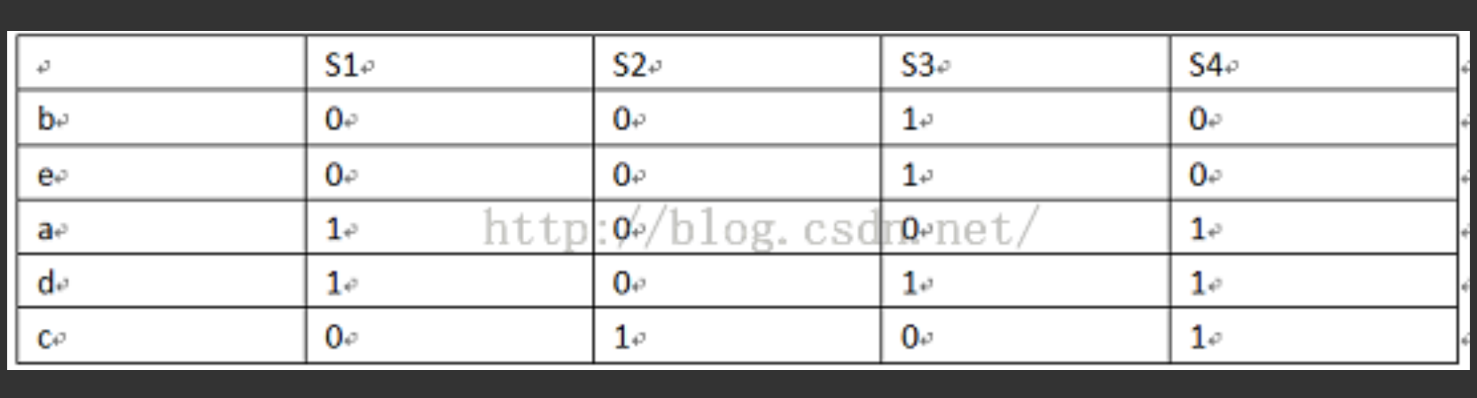
其中第一行和第一列不是特征矩阵的一部分，只是为了便于理解而写上的。

最小哈希

构建集合的特征矩阵是为了计算集合的最小哈希。

为了计算最小哈希，首先对特征矩阵的行进行打乱（也即随机调换行与行之间的位置），这个打乱是随机的。然后某一列的最小哈希值就等于打乱后的这一列第一个值为1的行所在的行号（不明白的直接看例子），行号从0开始。

例如，定义一个最小哈希函数h，然后对上面的特征矩阵进行行打乱，原来第一列的顺序为abcde，打乱后为beadc，则新的特征矩阵为：



	S1	S2	S3	S4
b	0	0	1	0
e	0	0	1	0
a	1	0	0	1
d	1	0	1	1
c	0	1	0	1

对于列S1，从这一列的第一行往下走，直到遇到第一个1，所在的行号则为这一列的最小哈希值。所以这4列的最小哈希值依次为h(S1) = 2, h(S2) = 4, h(S3) = 0, h(S4) = 2.

最小哈希与Jaccard相似度

在经过行打乱后的两个集合计算得到的最小哈希值相等的概率等于这两个集合的Jaccard相似度。简单推导如下：

假设只考虑集合S1和S2，那么这两列所在的行有下面三种类型：

1. 这一行的S1和S2的值都为1（即两列值都为1），记为X类；
2. 这一行只有一个值为1，另一个值为0，记为Y类；
3. 这一行两列的值都为0，记为Z类。

假设属于X类的行有x个，属于Y类的行有y个，所以S1和S2交集的元素个数为x，并集的元素个数为x+y，所以SIM(S1,S2) = x / (x+y)。注：SIM(S1,S2)就是集合S1和S2的Jaccard相似度。

接下来计算最小哈希h(S1) = h(S2)的概率。经过行打乱之后，对特征矩阵从上往下扫描，在碰到Y类行之前碰到X类行的概率是x / (x+y)；又因为X类行中h(S1)=h(S2)，所以h(S1)=h(S2)的概率为x/(x+y)，也就是这两个集合Jaccard相似度。

最小哈希签名

上面是用一个行打乱来处理特征矩阵，然后就可以得到每个集合最小哈希值，这样多个集合就会有多个最小哈希值，这些值就可以组成一列。当我们用多个随机行打乱（假设为n个，分别为h1,h2…hn）来处理特征矩阵时，然后分别计算打乱后的这n个矩阵的最小哈希值；这样，对于集合S，就会有n个最小哈希值，这n个哈希值就可以组成一个列向量，为[h1(S), h2(S)…hn(S)]；因此对于一个集合，经过上面的处理后，就能得到一个列向量；如果有m个集合，就会有m个列向量，每个列向量中有n个元素。把这m个列向量组成一个矩阵，这个矩阵就是特征矩阵的签名矩阵；这个签名矩阵的列数与特征矩阵相同，但行数为n，也即哈希函数的个数。通常来说，n都会比特征矩阵的行数要小很多，所以签名矩阵就会比特征矩阵小很多。

最小签名的计算

其实得到上面的签名矩阵之后，我们就可以用签名矩阵中列与列之间的相似度来计算集合间的Jaccard相似度了；但是这样会带来一个问题，就是当一个特征矩阵很大时（假设有上亿行），那么对其进行行打乱是非常耗时，更要命的是还要进行多次行打乱。为了解决这个问题，可以通过一些随机哈希函数来模拟行打乱的效果。具体做法如下:

假设我们要进行n次行打乱，则为了模拟这个效果，我们选用n个随机哈希函数h1,h2,h3…hn（注意，这里的h跟上面的h不是同一个哈希函数，只是为了方便，就不用其他字母了）。处理过程如下：

令SIG(i,c)表示签名矩阵中第i个哈希函数在第c列上的元素。开始时，将所有的SIG(i,c)初始化为Inf(无穷大)，然后对第r行进行如下处理：

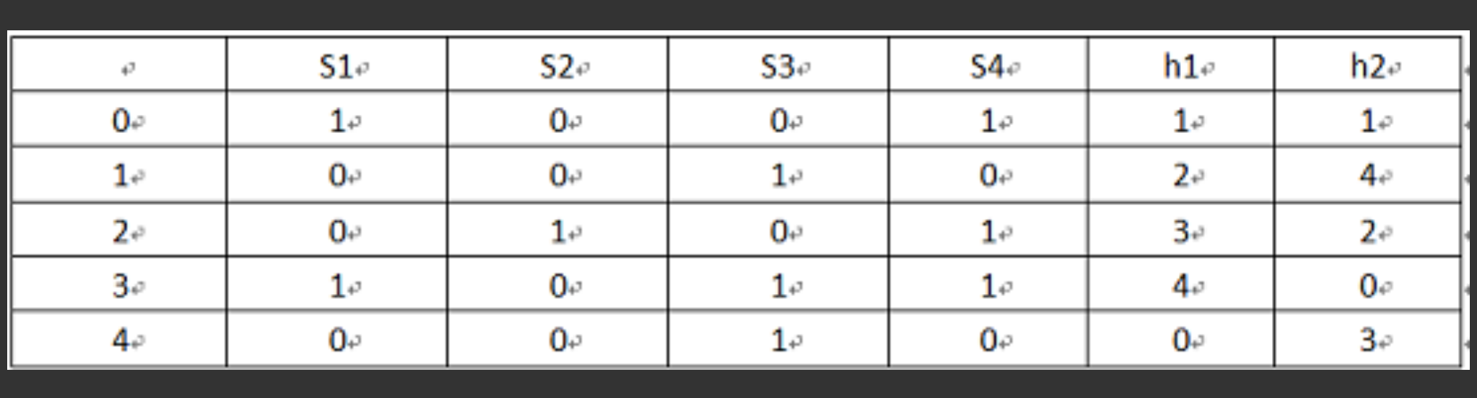
1. 计算h1(r), h2(r)…hn(r)；
2. 对于每一列c:

a) 如果c所在的第r行为0，则什么都不做；

b) 如果c所在的第r行为1，则对于每个i=1,2…n，将SIG(i,c)置为原来的SIG(i,c)和hi(r)之间的最小值。

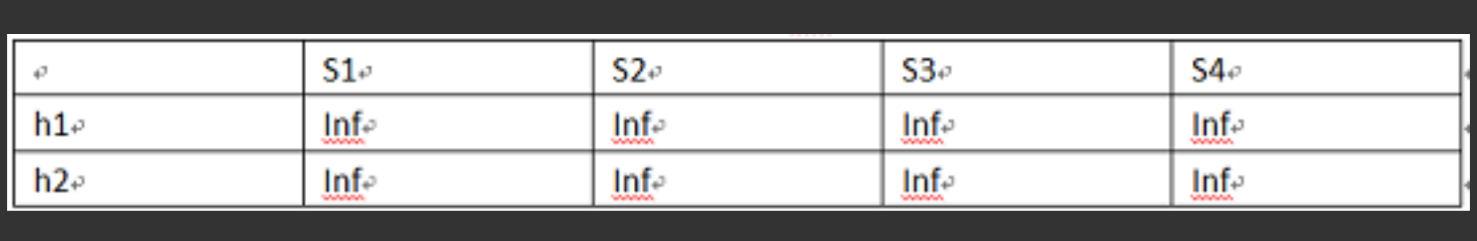
（看不懂的直接看例子吧，这里讲的比较晦涩）

例如，考虑上面的特征矩阵，将abcde换成对应的行号，在后面加上两个哈希函数，其中h1(x)=(x+1) mod 5，h2(x) = (3*x+1) mod 5，注意这里x指的是行号：



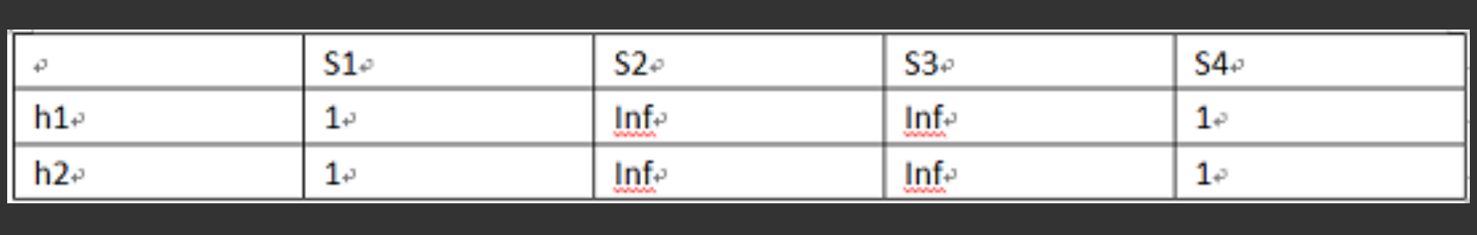
	S1	S2	S3	S4	h1	h2
0	1	0	0	1	1	1
1	0	0	1	0	2	4
2	0	1	0	1	3	2
3	1	0	1	1	4	0
4	0	0	1	0	0	3

接下来计算签名矩阵。一开始时，全部初始化为Inf:



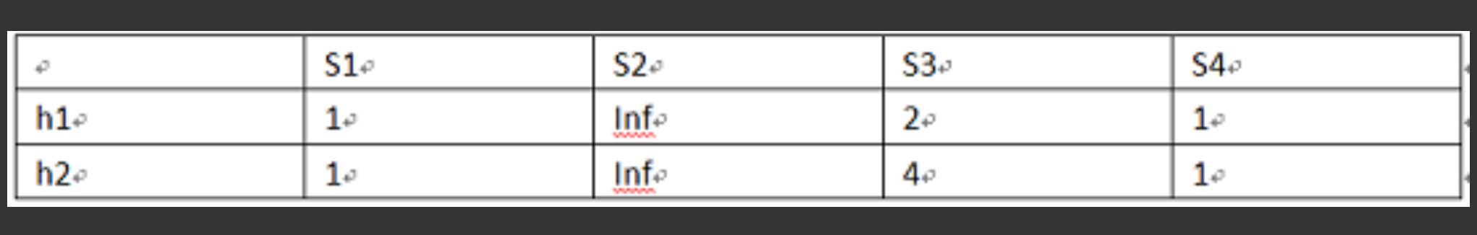
	S1	S2	S3	S4
h1	Inf	Inf	Inf	Inf
h2	Inf	Inf	Inf	Inf

接着看特征矩阵中的第0行；这时S2和S3的值为0，所以无需改动；S1和S4的值为1，需改动。h1= 1，h2= 1。1比Inf小，所以需把S1和S4这两个位置对应的值替换掉，替换后效果如下：



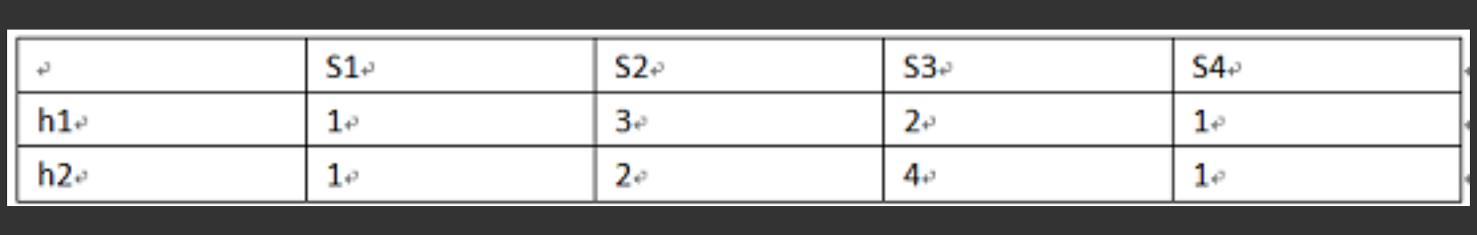
	S1	S2	S3	S4
h1	1	Inf	Inf	1
h2	1	Inf	Inf	1

接着看第1行；只有S3的值为1；此时h1= 2，h2= 4；对S3那一列进行替换，得到：



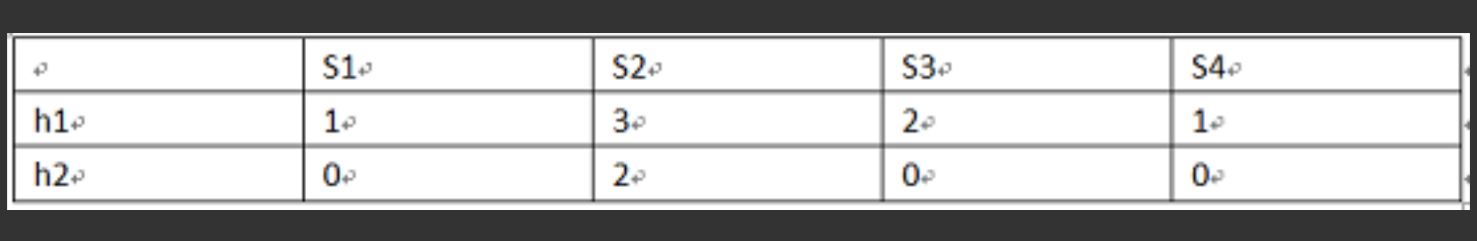
	S1	S2	S3	S4
h1	1	Inf	2	1
h2	1	Inf	4	1

接着看第2行；S2和S4的值为1；h1=3，h2=2；因为签名矩阵S4那一列的两个值都为1，比3和2小，所以只需替换S2那一列：



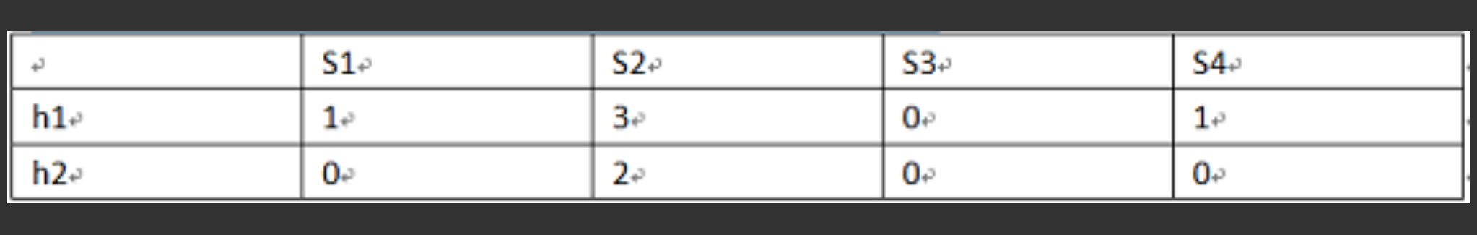
	S1	S2	S3	S4
h1	1	3	2	1
h2	1	2	4	1

接着看第3行；S1，S3和S4的值都为1，h1=4, h2= 0；替换后效果如下：



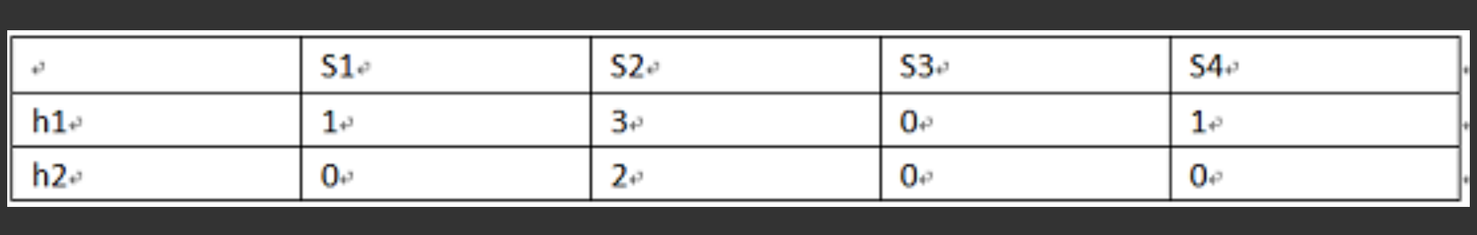
	S1	S2	S3	S4
h1	1	3	2	1
h2	0	2	0	0

接着看第4行；S3值为1，h1=0, h2= 3，最终效果如下：



	S1	S2	S3	S4
h1	1	3	0	1
h2	0	2	0	0

这样，所有的行都被遍历一次了，最终得到的签名矩阵如下：



	S1	S2	S3	S4
h1	1	3	0	1
h2	0	2	0	0

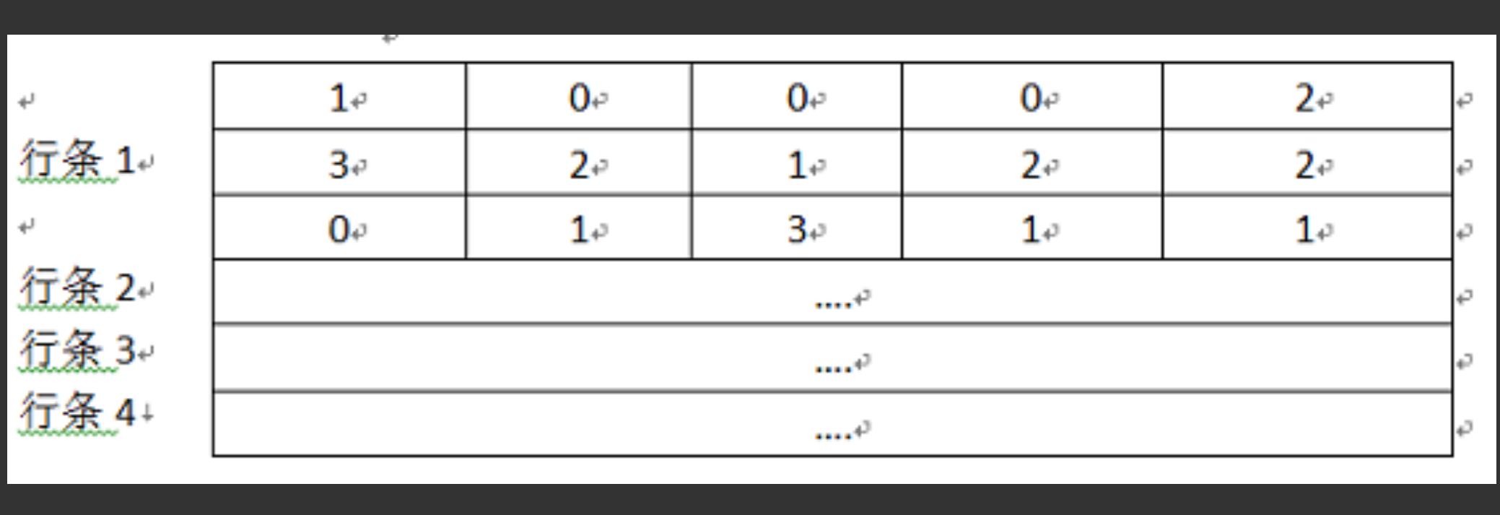
基于这个签名矩阵，我们就可以估计原始集合之间的Jaccard相似度了。由于S1和S4对应的列向量完全一样，所以可以估计SIM(S1，S4)=1;同理可得SIM(S1，S3) = 0.5;

局部敏感哈希算法（LSH）

通过上面的方法处理过后，一篇文档可以用一个很小的签名矩阵来表示，节省下很多内存空间；但是，还有一个问题没有解决，那就是如果有很多篇文档，那么如果要找出相似度很高的文档，其中一种办法就是先计算出所有文档的签名矩阵，然后依次两两比较签名矩阵的两两；这样做的缺点是当文档数量很多时，要比较的次数会非常大。那么我们可不可以只比较那些相似度可能会很高的文档，而直接忽略过那些相似度很低的文档。接下来我们就讨论这个问题的解决方法。

首先，我们可以通过上面的方法得到一个签名矩阵，然后把这个矩阵划分成b个行条(band)，每个行条由r行组成。对于每个行条，存在一个哈希函数能够将行条中的每r个整数组成的列向量（行条中的每一列）映射到某个桶中。可以对所有行条使用相同的哈希函数，但是对于每个行条我们都使用一个独立的桶数组，因此即便是不同行条中的相同列向量，也不会被哈希到同一个桶中。这样，只要两个集合在某个行条中有落在相同桶的两列，这两个集合就被认为可能相似度比较高，作为后续计算的候选对；下面直接看一个例子：

例如，现在有一个12行签名矩阵，把这个矩阵分为4个行条，每个行条有3行；为了方便，这里只写出行条1的内容。



	1	0	0	0	2
行条 1	3	2	1	2	2
	0	1	3	1	1
行条 2	....				
行条 3	....				
行条 4	....				

可以看出，行条1中第2列和第4列的内容都为[0,2,1]，所以这两列会落在行条1下的相同桶中，因此不过在剩下的3个行条中这两列是否有落在相同桶中，这两个集合都会成为候选对。在行条1中不相等的两列还有另外的3次机会成为候选对，因为他们只需在剩下的3个行条中有一次相等即可。

经过上面的处理后，我们就找出了相似度可能会很高的一些候选对，接下来我们只需对这些候选队进行比较就可以了，而直接忽略那些不是候选对的集合。